(c)1981

BASIC/S

The BASIC/S Compiler System

by: BILL STOCKWELL

Mod I or III - 48K required
and 1 disk drive. (2 preferable)

Published by PowerSoft
a div. of Breeze/QSD

Contains BASIC/S and BASICSII/CMD

Documentation by: Bill Stockwell
Final editing by: Renato Reyes, PHD
& Dennis A. Brent

First printing - March, 1982

-db

POWERSOFT

((tm)

11500 Stemmons Freeway, Suite 125 -- Dallas, Texas 75229

The BASIC/S Compiler System consists of two main programs - BASIC/S and BASICSII/CMD - along with numerous supplementary files. Both BASIC/S and BASIC/S II are compilers for a large subset of TRS-80 Disk BASIC - the first one, BASIC/S, is itself a BASIC program while BASICSII/CMD is a machine language version, compiled by BASCOM(c). The difference between them is that BASIC/S supports the full BASIC/S subset, while BASIC/S II is an integer compiler. It does not support floating point. Other than that, the two compilers support essentially the same BASIC subset. You get both compilers in one package. In general, one would want to use BASIC/S II (because of its speed), but when your application requires floating point, then BASIC/S is available.

Both compilers will run under virtually any Mod I/Mod III DOS, except TRSDOS Mod III. At least 48K and one disk drive are required to use BASIC/S. (Two drives are preferable).

Note : BASIC/S II does NOT run under Mod III TRSDOS due to the way the FCB is handled, (R. Shack's weirdness, not ours). For Model III, use ANY other DOS, (LDOS, NEWDOS/80, DOSPLUS, or Multidos).

It will compile up to a 260 line program - compiles into a /CMD file with no linking or run time module needed. No royalties are required for programs you write and compile with BASIC/S. A mention in the program and doc would be appreciated. ("Compiled by BASIC/S"). The /CMD files created by BASIC/S are very reasonably sized. Typically, they are only 1.2 - 2 times the size of your original BASIC source file. Quite often, if your source file is only 1 granule, then so is the /CMD file made by BASIC/S.

The name BASIC/S means BASIC/Subset. It does NOT compile full blown BASIC. It DOES support MOST of Level II Basic as well as the essential elements of sequential and random disk I/O, including LRL < 256. BASIC/S allows dimensioning arrays of all variable types, with up to two dimensions; any one program can have up to 20 arrays. Also, BASIC/S compiled programs can chain from one to another with NO loss of variables.

BASIC/S syntax is, in general, much more restrictive than regular Disk BASIC. Expressions need to be broken down to simple forms for the most part. Therefore, most programs will have to be rewritten to be compiled with BASIC/S. One area where the syntax is NOT so restrictive is in math expressions involving floating point variables (for BASIC/S, not BASIC/S II) - thus

```
A=7*SIN(X+Y*COS(A+B/C)-SQR(1/Z)*ARRAY(N%))
```

would be perfectly OK (just be sure to dimension ARRAY !)

Following is a list of the command/keywords/functions

supported by BASIC/S :

DEF FN	OPEN ("R","O","I","E")		LINE INPUT#	PRINT#
CLOSE	GET	PUT	FIELD	CLEAR
MKI	CVI	MKS	CVS	LSET
RND	RANDOM	CLS	LOF	IF
PRINT	PRINT@	LPRINT	INPUT	ASC
CHR$	VAL	STR$	LEFT$	RIGHT$
MID$ (both sides of =)		INSTR	INKEY$	LEN
GOTO	GOSUB	RETURN	CINT	CSNG
SET	RESET	POINT	PEEK	POKE
INP	OUT	AND	OR	FOR
NEXT	USR	DEFUSR	DATA	READ
RESTORE	RUN (as in RUN A$, A$=any dos command)			SIN
COS	TAN	ATN	EXP	LOG
ABS	SQR	INT	SET EOF	SCAN
HEX$	CMD	ON GOTO	ON ERROR	

STARTING OFF...

There are 2 diskettes contained in this package; both are self booting (on either a Mod I or Mod III) - just boot each disk in turn - it will display the files it has, prompt you for a destination drive, and dump the files to it. The destination drive must already be formatted (TRSDOS format, I or III). Single drive owners need to prepare 3 (yes, three!) disks ("stripped down&$$TRSDOW, with at least 50K free on each one). Boot the MASTER disks enclosed with this package, and follow the instructions on the screen. You will be prompted for the disk swaps.

Note : If you are using a Mod III, you are aware that BASIC/S is NOT compatible with Mod III TRSDOS. You still, however, need to "dump" the contents of your enclosed MASTER disks to a TRSDOS/III format. After the programs are on YOUR disks, you may "convert", "repair", or whatever operation the DOS you intend to use requires for accessing TRSDOS/III files.

Contents of the Disks:

Disk 1:
- BASIC/S — These two programs —
- COMPILER/DAT — go together. —
- QUICK/BAS
- REMPER/BAS
- BINHEX/BAS
- SQR/BAS
- SPACEWAR/BAS
- SHELL/BAS

CALC/BAS
COMPARE/BAS -the last four are for BASIC/S only-

Disk 2: BASICSII/CMD - these two programs -
COMPILE/DAT - go together -
FLOAT/TXT
FLOAT/BAS
FLOAT/CIM

For day to day use, you just need BASIC/S and COMPILER/DAT. (BASICSII/CMD and COMPILE/DAT in the case of BASIC/S II).

Each compiler has its own DAT file, which contains most of the data that is used to make up the /CMD files created by BASIC/S. Once BASIC/S is in memory and vunning, you can take the disk with BASIC/S on it out of the drive -- you only need the correct DAT file (and your source file) on line while compiling. The other files on the disks are supplementary files - mostly examples of BASIC/S compilable programs, to give you an idea of how to write BASIC/S code. By and large, it is not so different than writing any other BASIC program; you just have to watch your syntax more closely, and be aware that your program will be running in a compiled environment.

Following is the documentation for the two compilers, starting first with BASIC/S, and winding$up with BASIC/S II.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          BASIC/S COMPILER                             *
*            (C) 1981 by Bill Stockwell and Breeze/QSD                  *
*                 -Version 3.7 for Mod I and III-                       *
*                      -All Rights Reserved-                            *
*       Published by: Breeze/QSD, Inc., Dallas, Texas                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Getting Started Using BASIC/S:

IMPORTANT!! There is a variable in BASIC/S, in the very first line, which tells BASIC/S what disk operating system you are using. Currently, this is used so that LOF calculations will be done properly - ie when you compile a program that does an LOF calculation, it is important for the compiler to know what DOS is being used so that this calculation will be done properly. (The assumption is that you will run your /CMD files under the same system that they were compiled on. If this is not true, you need to change the variable as explained below and recompile under the other DOS).

The variable in question is KS, and is found at the end of line 1 of BASIC/S. It is now set as KS = 5. This is the correct setting for LDOS (tm).

Here are the other values :
Use KS=3 for Mod III TRSDOS
KS=4 for DOSPLUS 3.3 or earlier
and Ø for all other DOS's.

You can make this change and save BASIC/S with it, or you can specify KS when you RUN BASIC/S; when this is done, it overrides the KS setting in line 1. See the section on RUNNING the compiler for details.

On the disk you receive, there will be just one copy of BASIC/S, one of COMPILER/DAT, and some supplementary demo and utility files. Copy these to a disk of your own.

It is a good idea at this time to compile one of the sample programs on the disk. SHELL/BAS, LOOK/BAS, SPACEWAR/BAS, and COMPARE/BAS are all BASIC/S compilable. SHELL/BAS is a Shell Metzner sort program which will sort an ASCII sequential disk file of up to 79 strings; after you compile it, you invoke it via SHELL OUTPUT=INPUT from DOS READY mode, where INPUT is the file you want to

sort, and OUTPUT is a new file which you want the sorted file to be written to. Don't try to run SHELL/BAS from BASIC (as is). It checks the DOS command buffer at 4318H for the file specifications, which will not be meaningful from BASIC. TEST/DAT is a file for SHELL to sort. After compiling SHELL/BAS execute the /CMD file via SHELL OUT=TEST/DAT (where OUT is your output file). To compile SHELL/BAS, you should get into Basic and RUN"BASIC/S", making sure that COMPILER/DAT and SHELL/BAS are on line ; when BASIC/S asks "Files, options ?", respond with:

SHELL/BAS,SHELL/CMD,,56000 <enter>

This way, your command file will be placed into high memory, making room for a string array of dimension 79 (T$) in low memory. If you do not specify a starting address (which you normally wouldn't), it will default to 5200H, which means that the T$ array will be placed in high memory, where there is room for only 37 strings (and that's counting all the way up to FFFFH !). No problem if your file is no more than 37 strings long AND you have no high memory drivers in place; otherwise...
See below for more information on running BASIC/S.

COMPARE/BAS allows you to compare 2 files to see if they are the same.

SPACEWAR/BAS is a fast paced, real time shoot the Klingons game. You can run it in Basic or as a /CMD file after you compile it, but it runs MUCH faster compiled!

Also there is CALC/BAS, which is a DOS level calculator (after being compiled). See the remarks at the beginning of this program for more details.

For the most part, these programs allow you to become familiar with the rather restrictive syntax of BASIC/S.

The version of BASIC which is supported is a subset of Disk Basic. Only simple expressions and variable names are allowed, but most of the features and built-in functions of Level II are implemented, along with the essential elements of sequential and random disk I/O.

Note: Unlike regular BASIC, programs compiled by BASIC/S do NOT have any initialization of variables done. Thus numeric variables do not start out as zero, or strings as null. (See the CLEAR statement, however). One advantage of this approach is that one compiled program can invoke another (using the RUN statement) and all variables will be preserved.

Use of constants in BASIC/S is somewhat restricted; many statements allow (real or integer) constants ; most statements do NOT allow string constants. See the section below on the individual statements

for more details.

You may have multiple statements per line; the only restriction here is that IF, GOTO, and GOSUB statements (and ON GOTO statements, also) must begin the line they are on. Spacing is critical when writing a program to be compiled by BASIC/S; in general, use spaces only to separate keywords from identifiers (FOR N%=A% TO B% rather than FOR N%=A%TOB%).

Look over some of the sample programs on the disk to see how statements are to be coded. The syntax must be followed....

-------> E X A C T L Y !! <-------

The compiler allows the following variable names (all single letters): integers A% thru Z%, reals A-Z, and strings A$ thru Z$. Also, you may dimension arrays of any of these three types, and your array names can be any length, with every character significant. See the DIM statement for more on this.

REQUIREMENTS :

A TRS-80(c) Mod I or III with at least one drive and 48K. Repeat... 48K. Two drives are preferable.

RUNNING THE PROGRAM :

BASIC/S uses some USR routines (in the F000-F100 area), so you MUST set memory size at 61440 (X'F000') to run it. If you are using LDOS, you should set BLK=N as well. DOS+ requires you to specify the number of files; for BASIC/S, that number is 3. Also, use TBASIC when running under DOS+. Be sure that COMPILER/DAT and your program to be compiled (saved in ASCII) are on line when you run BASIC/S. REPEAT.... your program to compile MUST be saved in ASCII!

e.g. SAVE"FILENAME/BAS",A <enter>

(the /BAS extension is NOT required)

Now BASIC/S will ask:

Files, options ?

The typical response will be in the form:

SOURCE,OBJECT

e.g. TEST,TEST/CMD where TEST is the name of the ascii Basic program you want to compile, and TEST/CMD is the name of the load module you want to create. You may specify drive specs after either file name. It is best if the OBJECT file does not already exist. If it does exist, BASIC/S will kill it before continuing. No big deal, but it takes a little longer.

Three other parameters may be specified here. The first will produce output to the system line printer (in the form of source code and errors), while the second will tell BASIC/S where the load module's start point is to be. To specify line printer output, just put a * (or *PR, or *pr, or *anything) as the third parameter. The address, if present, should be a decimal integer in the fourth position. It may be positive or negative - Basic/s will respond correctly either way. Thus, complete syntax to the "Files ?" question is:

SOURCEFILE,OBJECTFILE,<*PR>,<ADDRESS>,<S=n>

Here the brackets "< >" are NOT to be typed, but indicate optional entries. If no printer output is wanted, but an address is to be specified, then the third parameter should be null - ie, present, but null or blank as indicated by two adjacent commas.

The S= sets KS, so that you may tell BASIC/S what DOS you are using at run time. Use S=3 for Mod III TRSDOS, S=4 for DOS+ 3.3 or earlier, S=5 for LDOS, and S=0 for all others. If the S= option is present, do not use nulls for the *PR or address options; if they are present, fine, but if not then leave them out. If *PR is not present, but address IS present, then leave an extra comma for the absent *PR.

Here are some examples :

```
TEST/BAS,TEST/CMD:1,,56000
       TEST,TEST/CMD:0,S=3
       TEST,TEST/CMD:0,,56000,S=0
```

The compiler gets much of the data needed to compile your program from a random access disk file (COMPILER/DAT). Be sure this file is on line when you use BASIC/S.

THE BASIC/S SUBSET (Statements supported under BASIC/S) :

PRINT

followed by a SINGLE variable name, or an expression in quotes. Thus :

```
PRINT A% or
PRINT"Message"
```

Also, you may use a semi-colon after anything being printed in order to suppress the carriage return.

PRINT@ is also supported -- just set any integer variable to the value of the location to be printed at, and you may then use any of the above forms with it. Thus :

```
PRINT@N%,"TRS-80";
```

LPRINT

Syntax for LPRINT is in every way the same as for PRINT, except of course that LPRINT@ has no meaning.

INPUT

You may input a single variable, of any type. You may not input a list of variables, but INPUT"PROMPT";A is supported (or A%, or A$). Note: Spacing is important in BASIC/S. Do not run keywords and variable names together -- use a single space in between them.
When executing a Basic/s compiled program, if input is requested, hitting the <break> key will cause an exit to DOS READY.
Also, if in answer to an input prompt, you hit <Enter> only, then the variable being inputted remains unchanged and the program continues (just like regular BASIC -- and this holds regardless of variable type (integer, real or string)).

LINE INPUT

LINE INPUT from the keyboard is supported. Syntax is exactly as it is in BASIC. You can even make LINEINPUT one word if you like. You may LINE INPUT a real or an integer variable if you wish, although this would not work in BASIC.

```
e.g.   LINE INPUT A$  or
       LINE INPUT "Prompt";A$   (just like in BASIC)
```

RUN A$

This statement allows you to set a string (A$ in this

case) to any DOS command, or the name of a command file you wish to invoke, and to exit the current program and have that command executed. Do NOT say RUN"PGM"; this will be not be correctly compiled! Also, RUN by itself is incorrect. The program being run, if compiled by BASIC/S, will NOT disturb the current values of BASIC/S variables. Thus you can chain from one BASIC/S compiled program to another with no loss of variables.

CLEAR

This statement, with or without an argument, will cause BASIC/S variables to be zeroed out. It depends on where your /CMD file starts; if your /CMD file is in low memory, then all memory from 41216 (decimal) up to HIGH$ will be zeroed out, while otherwise 5200H up to D6D8H is zeroed out. This makes sure that your /CMD file itself will never be affected, but that your variables will be zeroed. This works equally well on the Mod I or the Mod III - Basic/s knows which machine you are running it on, and will use the correct HIGH$ for your machine. DATA will also be cleared, and an automatic RESTORE done so that the DATA pointer will be correct.

GOTO ln

The GOTO statement. Do not space between the GO and the TO. DO space between the GOTO and the line number.

GOSUB ln

The standard GOSUB statement. Be sure your GOSUB's and RETURNS match up properly, or your /CMD file may crash.

DEF FN

This statement works almost exactly as in BASIC, the only limitations being that the right hand side must be already handleable by BASIC/S as in a normal assignment statement, and also only one argument is allowed. Thus it would be most useful in the case of the target variable being real, with the right hand side a real expression (see the section on assignment statements). But the argument and the target may be any type (real, integer, or string). Although only one argument is allowed, you may use any other variables you like on the right hand side -- but they won't be dummy.
Note : Constants may be used (real and integer,

anyway).

READ / DATA / RESTORE

Your program may have DATA statements, containing integer constants only (as in DATA 1,2,3) -- in all of your DATA statements you can have a total of 383 integers (no more). It is important that these DATA statements come before the READ statement(s) that are to access them (physically before, that is) -- the compiler generates code to place the data in memory when the DATA statements are encountered. Syntax for the READ statement is READ N% -- you can read only a single integer variable, which would normally be done in a FOR/TO loop. One big use for this is to poke DATA for a USR routine into memory. Before BASIC/S allowed READ/DATA, this process was rather clumsy.

RESTORE

works just like in standard BASIC.

IF

A very restricted IF statement -- you may only compare a floating point expression with zero, or two strings, or two simple integers (variables or constants). For floating points, syntax is:

```
IF X<Ø THEN 1ØØ
or   IF Z=Ø THEN 8Ø
or   IF SIN(A*B-C)<Ø THEN 2ØØ
(more on real expressions later).
```

The variable must be on the left. For strings, you can say

```
          IF A$<B$ THEN 2Ø
or        IF A$=B$ THEN 1ØØ
```

The compare must be in the '<' direction only, or with '='. You may check whether a string is null via

```
IF A$="" THEN 2ØØ (for example)
```

but this is the only time you may test a string against a constant.

For integers :

```
          IF A%=B% THEN 1ØØ
or        IF A%<B% THEN 5Ø
```

(and either A% or B% may be an integer constant, as in IF A%<72 THEN 200).

*** Note: GOTO, GOSUB, and IF statements MUST begin the line that they are on. Also, ELSE is now supported by BASIC/S; you may follow an IF statement with ELSE, and then as many statements as you like, as long as they aren't the type that must start the line they are on (IF, GOTO, GOSUB and ON GOTO). Thus :

```
IF EOF(1) THEN 200 ELSE LINE INPUT#1,A$:A$=A$+B$
```

FOR/NEXT

The For/Next loop is implemented for INTEGERS only. You may code

```
FOR A%=B% TO C%  (spacing important!)
    ...
NEXT A%
```

Constants may be used where B% and C% are indicated, as long as they are integers (positive, negative, or zero). Just be sure to use a single space after FOR and before and after TO. The variable in the NEXT statement is NOT optional. There is no STEP clause. FOR/NEXT loops may be (statically) nested.
The lack of the STEP clause is not a great problem; for example, to do FOR I%=5 TO 100 STEP 5, do this:

```
FOR I%=5 TO 100
    ...
I%=I%+4:NEXT I%
```

USR

A single USR call is allowed. It must be set up by DEFUSR, and the calling address must be a simple decimal integer constant. Thus :

```
DEFUSR=-1000
```

Note: There is no VARPTR statement. However, the addresses of all simple variables in BASIC/S are always the same and may be calculated as follows :

```
REALS   : If the ascii code for the variable is
          A, then the VARPTR will be
          -11535+4*(A-65).
INTEGERS : -11406+2*(A-65).
```

STRINGS : -23192+256*(A-65).

Strings are stored a little differently than in Level II. Each string is allocated 256 bytes, the first of which contains the length of the string (Ø to 255) and the rest of which contain the string itself. The Varptr points to the length byte.

Y%=USR(X%)

This causes the routine whose address was defined by a previous DEFUSR statement to be called. The current value of X% is loaded into the HL register pair before the call is made, and on return, Y% is given the value in the HL register pair. Do not use the ROM routines at ØA9A and ØA7F for this. Any integer variables may be used, not just X% and Y%. Also, a (decimal) integer constant may be used as the argument to be passed.

SET, RESET, and POINT

Use integers (either variables (followed by %) or constants) as the arguments. As with most BASIC/S functions, they may not be used in more complex expressions. Thus

```
SET(X%,2Ø)
A%=POINT(B%,C%)
```

The latter is the only way to access POINT - it cannot be invoked in an IF statement.

PEEK and POKE

Exactly as in Level II, except that the arguments must be integers -- (constants or variables). Thus

```
A%=PEEK(M%)
POKE A%,B%
POKE 15360,191
Z%=PEEK(14312)
```

INP and OUT

Syntax here is just like that for PEEK and POKE, i.e. you may use integer variables or constants as the arguments (no expressions).

A%=INP(P%) (input a byte from port P% and

```
                    store in A%)
OUT P%,V%  (output value V% to port P%)
OUT 255,1
S%=INP(232)
```

AND/OR

You may use these two functions in order to calculate an AND/OR result (for integer variables or constants) and store the answer in an integer variable. Thus

```
X%=Y% AND 20
U%=A% OR B%
```

CLS -- Clear the screen

RND

Random numbers between 0 and 1 may be generated by the statement X=RND(0). The left hand side may be any real variable. The argument is not actually required; you can simply say X=RND if you like. The statement RANDOM is also supported, to reseed the random number generator.

DIM

You can DIMension up to 20 arrays in a program to be compiled with BASIC/S - they can be integer, real or string, as distinguished by %, $, etc. The array names may be any length (up to 255) with every character significant.

ONLY letters A-Z should be used for the array names. (Actually, any characters except digits 0-9 may be used, although you should avoid $ and % as they are used to determine variable type).
Thus

```
DIM ARRAY(20,7),ST$(15),NUM%(50)
```

You may have one or two dimensions for each array - no more. DO NOT use BASIC keywords in your array names. Be careful about your available array space - BASIC/S will tell you if your array space will overlay BASIC/S data areas or the currently set high memory. It will also let you know exactly where your array space lies- if the latter number is FFFF, look out! That means that your arrays are dimensioned too large (almost certainly).

If this happens, try recompiling with a start address of 56000; this will give you about 19.75 K of space for your arrays, as it puts your /CMD file in high memory instead of low.
Still, 19.75K is only enough room for a string array of dimension 79 (79 * 256 = 20,224). With real and integer arrays, you can use much larger dimensions.

Syntax for using array elements :
For the most part, you can use your array variables just like any other variables; and you may always use integer constants (as well as variables) for the subscript(s).
Thus

```
READ NUM%(I%)
INPUT ARRAY(7)
PRINT ST$(U%);
A$=LEFT$(ST$(5),NUM%(I%))
```

The exceptions are as follows :
When an array element is on the left hand side of the '=' sign, the right hand side MUST be a simple variable or constant (string or numeric) - no expressions (or more array elements) allowed.

Also, any statement that references an array element should contain NO numeric constants of any kind, except for (possibly) subscripts to the array itself.
One exception here is that array elements may be compared via the IF statement, and the line number reference will not be misconstrued. So

```
IF ST$(1)<ST$(I%) THEN 75
```

is OK; just be sure to follow the syntax in all other respects. But something like

```
LINE INPUT#1,ST$(I%)
```

or

```
PUT 1,L%(I%)
```

won't work as the '1' will be misunderstood, and translated to a temporary integer variable, which won't work.

SCAN

This non standard statement, not found in regular BASIC, allows you to read a file or device a byte at a time, similar to INKEY$. Syntax is

```
SCAN b,A$
```

where b=DCB number, and the byte read (if any) is stored in A$. The file or device must be opened first.

HEX$

This statement is for hex conversion; it takes an integer argument, and converts it to its hex string equivalent. So if N%=255, then A$=HEX$(N%) would give A$ the value "FF".

SET EOF

This statement is for use with LDOS only; it allows you to truncate a random access file. Under BASIC/S, random access files are limited to DCB numbers 1 or 2; so the syntax is SET EOF1 or SET EOF2. To do it, you GET or PUT the last record in the file that you want it to have, and then SET EOF and CLOSE it to chop it off. For example, if you wanted to chop a file off at 50 records :

```
R%=50:GET 1,R%:SET EOF1:CLOSE 1
```

CMD

Syntax here is just CMD A$, where A$ is any string variable containing a command you wish to pass to DOS and return from. Typically, you would compile a program that uses CMD at 56000, or at least above the DOS command area 5200-6FFF (hex); otherwise your /CMD file will likely be overwritten. Do not use CMD "xxxx"; set a string variable to the command you want.

ON ERROR

BASIC/S supports a limited form of error trapping; you can trap DOS errors only using it. First, your ON ERROR statement must appear AFTER the error trap routine itself. Thus your program would typically start out branching around the error routine, to the ON ERROR statement. Thus:

```
50 ON ERROR GOTO 100
```

would not be legal, since line 100 comes after 50. Secondly, while ERR is supported, ERL and RESUME are not.

It is very important that the very FIRST thing your error

trap routine does is A%=ERR (ie, set some integer variable equal to ERR). If you wait to do this, ERR will change and not be relevant. Also, this is the only legal use of ERR; you must set an integer equal to it, period. ERR will contain the DOS error code that was detected by the DOS and returned in the A register; consult your DOS manual to see which codes refer to which errors. A code of 24, for example, means File Not Found. This does not trap such things as division by zero or illegal function call; only disk errors are covered here.

ASSIGNMENT statement :

Following are the allowed forms of the assignment statement.

REAL :

```
X=Y         (any var=any other var)
X=const     (var=constant value)
X=-Y        (var=-other var)
```

X=real expression

Here (and in the IF statement for real expressions) is the only place where BASIC/S can handle complex expressions. A "real expression" is defined as any combination of the real variables A-Z, +,-,*,/, ,(,), and the built-in functions SIN, COS, INT, TAN, ATN,LOG, EXP, SQR, and ABS, and up to 4 constants. Thus :

```
Y=5*SQR(Z*SIN(2*X+C)), for example.
```

Be careful with constants - you may only have 4 "active" constants at one time (for each var type), and this includes not only obvious constants, but also unary minus signs - thus Z=2*(-X) would have two constants (it would be translated into 2*(0-X)). Important note -- if you divide by a product, be careful. BASIC/S will interpret A/B*C as A/(B*C) rather than the usual (A/B)*C. This is due to the right to left parsing algorithm that is used. Use parantheses if in doubt. Another point is this: If you calculate X Y (X to the Yth power), this is done a little differently than in Level II -- it is calculated as EXP(Y*LOG(X)). Since LOG(X) is undefined for X<=0, this will CRASH your BASIC/S /CMD file, whereas BASIC will normally handle it if it makes sense as a real

number. So if you want to do such a calculation, you should check for X being 0 or negative.

```
X=CSNG(X%)
```

You can use this function to convert integers to real, but BASIC/S supports use of integer variables in real expressions, so it is rarely needed. (You may NOT use a real variable in an integer expression, though). Thus, X%=CINT(X) is a needed function, for converting reals to integers; and this function may be freely used wherever an integer variable would normally be expected. Thus SET(CINT(X),CINT(Y)) or GET 1,CINT(R) would be fine.

INTEGERS :

Integer arithmetic is limited to +,-,* and only 2 operands allowed on the right hand side. No builtin functions for integers. Constants may be used, however. Thus:

```
X%=A%*B%
X%=5-B%
```

Note that unary minus is not allowed here (for variables) ie X%=-Y%+Z% is no good, while X%=Z%-Y% is OK. Of course you may use unary minus if the right hand side is a single variable, as in X%=-Y%.

STRINGS :

```
A$=B$
A$="constant"
A$=B$+C$      (simple concatenation)
```

Also we have the builtin string functions ASC, LEN, CHR$, LEFT$, VAL, RIGHT$, MID$, STR$, and INSTR. Where numeric arguments are required in the string functions, simple integer variables or constants must be used - no expressions. The actual string arguments cannot be constants, but

```
A$=LEFT$(X$,2)
```

(for example) would be OK.

Also, expressions must be reduced to their simplest form -- e.g., concatenation within a function or function composition is not allowed. Break it down! Note: The INSTR function differs from the regular DISK BASIC one in that no starting position may be

specified -- syntax is just N%=INSTR(A$,B$). However, unlike previous versions of BASIC/S, ALL of B$ is searched for, not just the first character.
MID$ note -- you can use MID$ on the left hand side of the = sign, and in that case, you can use either of the two forms MID$(A$,N%)=B$ or MID$(A$,N%,L%)=B$ -- but they will give the same results, i.e. the length of B$ is used, L% is ignored in the second form. If the source string (B$) is null, nothing is done.
Note III: The INKEY$ function is implemented, and must be used in the form: A$=INKEY$ (or B$, etc.). Also, VAL may be used for reals only; i.e.,

```
X=VAL(A$)
```

and conversely, STR$ works only on floating point variables (A$=STR$(Y), for example). Do NOT use a constant as an argument to STR$.

DISK I/O statements

Essentially, you have ten disk I/O buffers available for use (0-9), all of which may be used for sequential access, and two (1 and 2) of which may be used for random access. Here are the specifics :

OPEN

The OPEN statement is essentially that of disk BASIC, except that the filespec must be a string variable and not an expression in quotes. Syntax is

```
        OPEN"m",b,F$<,r>
   where m  = mode = I,O,R, or E
         b  = buffer = (0-9)  (constant only)
              (must be 1 or 2 for direct access)
         F$ = filespec (variable only)
         r  = logical record length (optional -- may be
              either  an integer constant or an integer
              variable).
```

BASIC/S makes few restrictions on your use of the disk I/O statements, so be careful. For example, if you wanted to open a sequential file with an LRECL of 16, you could. However, you would probably be well advised to stick to direct access files for this!

OPEN"E" is like OPEN"O" except you start out positioned at the end of the file.

Sequential I/O is done with the LINE INPUT# and PRINT# statements. Just specify a buffer number adjacent to the #, and you are ready to go. Only a simple string variable may be input or output, although PRINT#1,A$; will disable the carriage return.

Random disk I/O is accomplished via the following :

FIELD

You must field your buffer in order to communicate between your strings and the disk file being accessed. Syntax is :

FIELD 1,nn AS A$,mm AS B$, ...

-- the buffer can be 1 or 2, the strings can be any of A$ thru Z$ (no array references allowed here!), and the numbers 'nn', 'mm' etc must be integer constants (1-255 -- 0 is not allowed). Also you can't really use multiple FIELD statements for the same file.
-- the second will override the first. Moreover, the statements to process a random access file must be statically nested -- i.e. do not GOSUB or GOTO a later line to FIELD a buffer and then return to do your LSETs and PUTs, etc. Just OPEN the file, FIELD the buffer, process it, and CLOSE it, without GOSUBS and GOTOS. (At least, don't branch anywhere outside the range of statements between the OPEN and CLOSE stmts).

LSET

To place your strings into the buffer prior to being PUT to the disk, use LSET. Thus

LSET A$=B$ (spacing critical!)

where A$ is one of the strings mentioned in your FIELD statement. If LEN(B$) is less than that of the field variable A$, it will be filled out with spaces in the buffer. If greater, only the leftmost portion of B$ (for the fielding length of A$) will be in the buffer.

PUT

Syntax is PUT b,N% where b is the buffer number (1 or 2) and N% is any integer variable, containing the record number to be put. The record number variable is not optional.

GET

As in GET 1,R% -- gets the R%th record from the disk file, and places its contents into the string variables mentioned in the FIELD statement.

LOF

The LOF function is implemented and syntax is

N%=LOF(b)

where b is the buffer number (1 or 2 -- must be a constant). This returns the number of records in the currently open file with buffer b. The setting of the BASIC/S variable KS is critical when using LOF; be sure it is set to the correct value for your DOS (3 for Mod III TRSDOS, 4 for DOS+ 3.3 or earlier, 5 for LDOS, and 0 for all others).

CVI and MKI$

For convenience in reading and writing integers from/to direct access files, these functions are implemented as in TRSDOS. In case you were mystified as to exactly what they did -- well, if the integer N% has the 2 byte representation (L,H), then MKI$(N%) is just CHR$(L)+CHR$(H). CVI just does the exact reverse. As with most BASIC/S functions, these may be used only with simple integer/string variables.

Also implemented (completely similarly) are CVS and MKS$. Since BASIC/S doesn't support double precision, CVD and MKD$ are not implemented.

CLOSE

There is no global close in BASIC/S -- you must mention the buffer number. Thus,

CLOSE 5

would close the file with buffer number 5. If you close

a file that isn't open, you will bomb out with 'FILE NOT OPEN'.

EOF

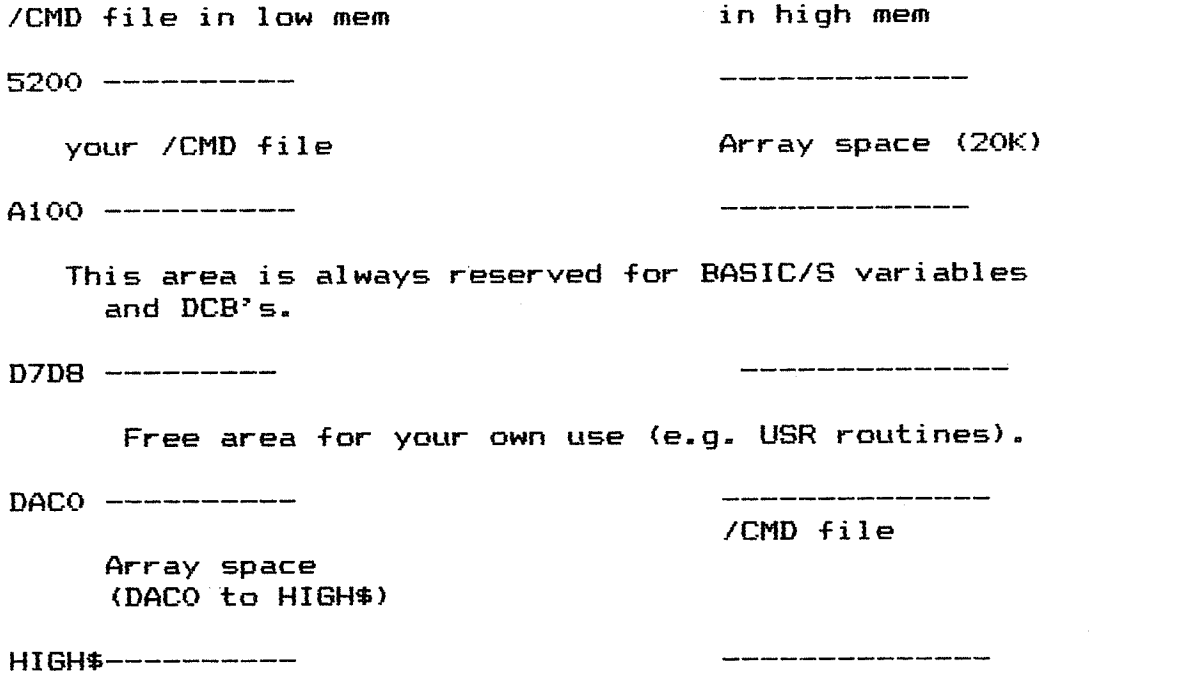
This isn't a function as such; it is to be used in a special form of the IF statement to check for EOF when inputting from a sequential file. Simply say

IF EOF(b) THEN 200

(or whatever line number) to check for end of file on buffer b (0-9)

BASIC/S Memory Map

Following is a map of memory from 5200H up to HIGH$, showing how BASIC/S uses the memory in your TRS-80 (48K):



==========

--DISCLAIMER OF WARRANTIES & LIMITATIONS OF LIABILITIES --

We have taken great care in preparing this package. We make no expressed or implied warranty of any kind with regard to this manual or to BASIC/S. In NO event shall we be liable for incidental or consequential damage in connection with or arising out of the performance of this program.

Questions should be addressed to:

Bill Stockwell
4771 NW 24th #228N
Oklahoma City OK 73127
(405) 947-4156
Mnet 70070,320

Bill Stockwell may also be reached on the QSD Sig on MicroNet. Leave a message to 70001,610 for info or from the OK prompt, type R QSD<enter>.

Published by:
PowerSoft - a division of Breeze/QSD, Inc.
11500 Stemmons Expressway Suite 125
Dallas, Texas 75229

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          BASIC/S II COMPILER                          *
*            (C) 1982 by Bill Stockwell and Breeze/QSD                  *
*                 -Version 1.5 for Mod I and III-                       *
*                       -All Rights Reserved-                           *
*        Published by: Breeze/QSD, Inc., Dallas, Texas                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Getting Started Using BASIC/S II:

IMPORTANT!! There is a variable in BASIC/S which tells BASIC/S what disk operating system you are using. Currently, this is used so that LOF calculations will be done properly - ie when you compile a program that does an LOF calculation, it is important for the compiler to know what DOS is being used so that this calculation will be done properly. (The assumption is that you will run your /CMD files under the same system that they were compiled on. If this is not true, you can change the value of this variable as explained below and recompile under the other DOS). This is discussed under "Options", when you execute BASICSII.

On the disk you receive, there will be just one copy of BASICSII/CMD, one of COMPILE/DAT, and some supplementary demo and utility files. Copy these to a disk of your own. One of these files is REMPER/BAS, a utility useful for those who have programs written for the original BASIC/S (the original BASIC/S requires percent signs after integer variable names, whereas BASICSII regards A-Z as integer variables - no percent signs allowed!) REMPER will remove all percent signs from an ascii BASIC file, so that it will now be compilable by BASICSII so long as no real variables are used in it.

Another file is QUICK/BAS, which generates an integer array and does a quicksort on it, and prints out the results.

Also there is BINHEX/BAS, originally by Tim Mann and rewritten by Bill Stockwell into BASIC/S compilable form. This program is for converting HEX files to/from /CMD file format. HEX files are the typical way in which binary files are stored on bulletin board systems for transfer via modem.

Finally on the disk is a utility for allowing BASICSII to handle floating point values (in a limited way) (see the files FLOAT/TXT, FLOAT/BAS, and SQR/BAS for more info on this).

The version of BASIC which is supported is a subset of Disk Basic. Only simple expressions and variable names are allowed, but most of the features and built-in functions of Level II are implemented, along with the essential elements of sequential and random disk I/O. Floating point variables are not supported (BASICSII/CMD is 40K as it is!), but integers, strings, and arrays of

type integer or string are allowed. Note: Unlike regular BASIC, programs compiled by BASIC/S do NOT have any initialization of variables done. Thus numeric variables do not start out as zero, or strings as null. (See the CLEAR statement, however). One advantage of this approach is that one compiled program can invoke another (using the RUN statement) and all variables will be preserved.

Use of constants in BASIC/S is somewhat restricted; many statements allow (integer) constants ; most statements do NOT allow string constansts. See the section below on the individual statements for more details.

You may have multiple statements per line; the only restriction here is that IF, GOTO, and GOSUB statements must begin the line they are on (as must ON GOTO).

Look over some of the sample programs on the disk to see how statements are to be coded. The syntax must be followed....

-------> E X A C T L Y !! <-------

...however, the spacing is up to you. Thus, you could say FOR I=1 TO N or you could say FORI=1TON. The compiler allows the following variable names (all single letters): integers A thru Z and strings A$ thru Z$. Also, you may dimension arrays (of either type), and your array names can be any length, with every character significant. See the DIM statement for more on this.

REQUIREMENTS :

A TRS-80(c) Mod I or III with at least one drive and 48K.
Repeat... 48K! Two drives are certainly preferable.

RUNNING THE PROGRAM :

Just type BASICSII from Dos Ready. Be sure that you do not have too much in high memory -- if HIGH$ is less than FE00H, the compiler may run out of string space (or other strange errors may occur). If you are running Mod I LDOS, you can run Lower Case and PDUBL, but not much else in high memory. For best results, run with HIGH$ (HIMEM) = FFFF, if possible.
(Note on LDOS 5.1 for Mod I - you may have PDUBL and KI/DVR in high memory - nothing else - to use BASIC/S II. When you SET KI/DVR, do NOT use type ahead).
After BASICSII begins, you may remove the disk containing it, and insert the one with COMPILE/DAT, if necessary, (if you have only one drive, this must be a SYSTEM disk). Remember -- COMPILE/DAT must be

on line AT ALL TIMES while you compile, as well as the program you wish to compile (saved in ASCII!).

REPEAT.... your program to compile MUST be saved in ASCII!

e.g. SAVE"BASPROGM/BAS",A <enter>

(the /BAS extension is NOT required)

Now BASIC/S will ask:

```
Source :
Object :
Options :
```

(one after another). Typically, you will answer the first two questions, and hit <enter> on the last one. "Source" is the name of the file to be compiled, and "Object" is the name of the /CMD file you want to create from "Source". Thus if you had an ASCII BASIC program called TEST/BAS that you wanted to compile, you might answer the above with:

```
Source : TEST/BAS
Object : TEST/CMD:2
Options : <enter>
```

The options you might take are as follows :

You can specify Start Address, whether or not to list the source file to the video during the compile, whether to disable the <break> key (for the compiled code), and what DOS is being used. Any or all may be specified, but those that are used should be in the correct order. The start address tells BASICSII where in memory your object file should load to -- the default being 5200H (20992 decimal). The address MUST be a decimal integer, but can be positive or negative; ie D6D8H is represented by either 55000 or -10536; BASIC/S knows what you mean.

Use the letter N to indicate No list - BASIC/S normally lists your source file to the video as it compiles, but if you don't want this, just answer Options : with 'N' (after the address, if there is one). You might want to do this if you were getting a lot of errors which were scrolling off the screen too fast.

You can specify the DOS you are using via:
S=x (x being an integer value).

Use:
4 for DOSPLUS 3.3
5 for LDOS
1 for Newdos/80 and DOS+ 3.4
0 for any other DOS
The default is 5 (LDOS).

TRSDOS (Mod III) is _N_O_T__S_U_P_P_O_R_T_E_D.

To start the /CMD file at 56000, with no listing, and using Mod I TRSDOS, you would answer 'Options' with

Options : 56000,N,S=0

Note that the S= option is important ONLY if your source file computes LOF's. Also note that BASIC/S accepts lower case responses.

A new option, added in version 1.1 of BASIC/S II, allows you to disable the break key while the BASIC/S /CMD file that is created executes. This is done by typing the letter "B" among your options - like the other options, this must come AFTER any address. Doing this will cause the following to occur: during input, while a BASIC/S /CMD file executes with the "B" option, if the <break> key is typed, then a line feed is done and the input starts over at the beginning.

THE BASIC/S SUBSET (Statements supported under BASIC/S) :

PRINT

followed by a SINGLE variable name, or an expression in quotes. Thus :

```
PRINT A or
PRINT"Message" or
PRINT R$
```

Also, you may use a semi-colon after anything being printed in order to suppress the carriage return.

PRINT@ is also supported -- just set any integer variable (or constant) to the value of the location to be printed at, and you may then use any of the above forms with it. Thus :

```
PRINT@N,"TRS-80";
```

It is important to note that you may NOT print a list of items when using BASIC/S; only one item (of any type) may be PRINTed at a time. The same applies to INPUT.

LPRINT

Syntax for LPRINT is in every way the same as for PRINT, except of course that LPRINT@ has no meaning.

INPUT

You may input a single variable, of any type. You may not input a list of variables, but INPUT"PROMPT";A is supported (or A$).
When executing a Basic/s compiled program, if input is requested, hitting the <break> key will cause an exit
If in answer to an input prompt, you hit <Enter> only, then the variable being inputted remains unchanged and the program continues (just like regular BASIC -- and this holds regardless of variable type.

LINE INPUT

LINE INPUT from the keyboard is supported. Syntax is exactly as it is in BASIC. You may LINE INPUT an integer variable if you wish, although this would not work in BASIC.

e.g. LINE INPUT A$ or
LINE INPUT "Prompt";A$ (just like in BASIC)

RUN A$

This statement allows you to set a string (A$ in this case) to any DOS command, or the name of a command file you wish to invoke, and to exit the current program and have that command executed. Do NOT say RUN"PGM"; this will be not be correctly compiled! Also, RUN by itself is incorrect.

DEFINT

For compatibility with the BASIC interpreter, you may use this statement (as in DEFINT A-Z). No other DEF statements are accepted, and this one only reaffirms what BASIC/S II does anyway - regards all variables as integers unless they are suffixed with '$'.

CLEAR

This statement, with or without an argument, will cause BASIC/S variables to be zeroed out. It depends on where your /CMD file starts; if your /CMD file is in low memory, then all memory from 41216 (decimal) up to HIGH$ will be zeroed out, while otherwise 5200H up to D6D8H is zeroed out. This makes sure that your /CMD file itself will never be affected, but that your variables will be zeroed. This works equally well on

the Mod I or the Mod III - BASIC/S knows which machine you are running it on, and will use the correct HIGH$ for your machine. DATA will also be cleared, and an automatic RESTORE done so that the DATA pointer will be correct.

GOTO ln

The GOTO statement -- works just as in BASIC, but it MUST BEGIN the line it is on. Thus CLS:GOTO 20 will not work, although no message would be given.

ON GOTO

As in BASIC, except that the index must be a simple variable (not an expression). Thus

```
ON X GOTO 20,30,1000
```

No limit on the number of different lines you can branch to, other than the limitation of 255 chars per line. ON GOSUB is NOT supported. Like IF, GOTO, and GOSUB, ON GOTO statements must begin the line they are on.

GOSUB ln

The standard GOSUB statement, but like GOTO, must begin the line it is on.

READ / DATA / RESTORE

Your program may have DATA statements, containing integer constants only (as in DATA 1,2,3) -- in all of your DATA statements you can have a total of 383 integers (no more). It is important that these DATA statements come before the READ statement(s) that are to access them (physically before, that is) -- the compiler generates code to place the data in memory when the DATA statements are encountered. Syntax for the READ statement is READ N -- you can read only a single integer variable, which would normally be done in a FOR/TO loop. One big use for this is to poke DATA for a USR routine into memory. Before BASIC/S allowed READ/DATA, this process was rather clumsy.

RESTORE

works just like in standard BASIC.

IF

A very restricted IF statement -- you may only compare two strings (for equality or in the < direction), or two simple integers (variables or constants).
Thus (for strings) :

```
IF A$<B$ THEN 20
or      IF A$=B$ THEN 100
```

The compare must be in the '<' direction only, or with '='. You may check whether a string is null via

```
IF A$="" THEN 200 (for example)
```

but this is the only time you may test a string against a constant.

For integers :

```
IF A=B THEN 100
or      IF A<B THEN 50
```

(and either A or B may be an integer constant, as in IF A<72 THEN 200).

*** Note: GOTO, GOSUB, and IF statements MUST begin the line that they are on. Also, ELSE is now supported:
you may follow any IF statement with ELSE, followed by as many statements as you can fit on one line, so long as they do not need to start the line they are on. Thus IF, GOTO, GOSUB, and ON GOTO statements may not follow an ELSE, but any other statement may do so.

FOR/NEXT

The For/Next loop is implemented for INTEGERS only. You may code

```
FOR A=B TO C
    ...
NEXT A
```

Constants may be used where B and C are indicated, as long as they are integers (positive, negative, or zero).
The variable in the NEXT statement is NOT optional.
There is no STEP clause.
FOR/NEXT loops may be (statically) nested.

USR

A single USR call is allowed. It must be set up by

DEFUSR, and the calling address must be a simple decimal integer constant. Thus :

```
DEFUSR=-1000
```

Note: There is no VARPTR statement. However, the addresses of all simple variables in BASIC/S are always the same and may be calculated as follows :

If A is the ascii code of the variable in question, then the VARPTR is :

```
INTEGERS : -11406 + 2 * (A - 65);
STRINGS  : -23192 + 256 * (A - 65).
```

Strings are stored a little differently than in Level II. Each string is allocated 256 bytes, the first of which contains the length of the string (0 to 255) and the rest of which contain the string itself. The Varptr points to the length byte.

Y=USR(X)

This causes the routine whose address was defined by a previous DEFUSR statement to be called. The current value of X is loaded into the HL register pair before the call is made, and on return, Y is given the value in the HL register pair. Do NOT call the ROM routines at 0A7F and 0A9A for this. Any integer variables may be used, not just X and Y. Also, a (decimal) integer constant may be used as the argument to be passed.

SET, RESET, and POINT

Use integers (either variables (followed by) or constants) as the arguments. As with most BASIC/S functions, they may not be used in more complex expressions. Thus

```
SET(X,20)
A=POINT(B,C)
```

The latter is the only way to access POINT - it cannot be invoked in an IF statement.

PEEK and POKE

Exactly as in Level II, except that the arguments must be integers -- (constants or variables). Thus

```
A=PEEK(M)
```

```
POKE A,B
POKE 15360,191
Z=PEEK(14312)
```

INP and OUT

Syntax here is just like that for PEEK and POKE, i.e. you may use integer variables or constants as the arguments (no expressions).

```
A=INP(P) (input a byte from port P and
            store in A)
OUT P,V  (output value V to port P)
OUT 255,1
S=INP(232)
```

AND/OR

You may use these two functions in order to calculate an AND/OR result (for integer variables or constants) and store the answer in an integer variable. Thus

```
X=Y AND 20
U=A OR B
```

CLS -- Clear the screen

RND

Random integers between 1 and N may be generated by the statement X=RND(N). The left hand side may be any integer variable. The argument is required and may be an integer constant if you like. The statement RANDOM is also supported, to reseed the random number generator.

DIM

You can DIMension up to 20 arrays in a program to be compiled with BASIC/S - they can be either integer or string,as distinguished by the presence of a $.
Array names may be any length (up to 255) with every character significant. ONLY letters A-Z should be used within an array name. Thus

```
DIM ARRAY(20,7),ST$(15)
```

You may have one or two dimensions for each array - no

more. DO NOT use BASIC keywords in your array names. Be careful about your available array space - BASIC/S will tell you if your array space will overlay BASIC/S data areas or will exceed the 64K memory limit. If this happens, try recompiling with a start address of 56000; this will give you about 19.75 K of space for your arrays, as it puts your /CMD file in high memory instead of low. Still, 19.75K is only enough room for a string array of dimension 79 (79 * 256 = 20,224). With integer arrays, you can use much larger dimensions.

Syntax for using array elements :
For the most part, you can use your array variables just like any other variables; and you may always use integer constants (as well as variables) for the subscripts). Thus

```
READ NUM(I)
INPUT ARRAY(7)
PRINT ST$(U);
A$=LEFT$(ST$(5),NUM(I))
```

The exceptions are as follows :
When an array element is on the left hand side of the '=' sign, the right hand side MUST be a simple variable or constant of the same type - no expressions allowed. Thus ST$(1)=LEFT$(A$,2)is not allowed; you would need to set H$=LEFT$(A$,2) and then ST$(1)=H$. However, it is OK to set an array element to a constant, as in ST$(5)="HELLO" or ARRAY(14,6)=12.
Also, any statement that references an array element should contain NO numeric constants of any kind, except for (possibly) subscripts to the array itself.
One exception here is that array elements may be compared via the IF statement, and the line number reference will not be misconstrued. So

```
IF ST$(1)<ST$(I) THEN 75
```

is OK; just be sure to follow the syntax in all other respects. But something like

```
LINE INPUT#1,ST$(I)
```

or `PUT 1,L(I)`
won't work as the '1' will be misunderstood, and translated to a temporary integer variable, which won't work.
Thus in general, the statements in which you may not reference array elements are most of the DISK I/O statements (OPEN, FIELD, GET, PUT, LINE INPUT#, PRINT#), and PRINT@.

SCAN

This statement allows the user to "scan" a file or device for a single byte (similar to INKEY$ for the keyboard). First you OPEN the

file or device in question for input; then

```
SCAN b,A$
```

will read a byte from the file or device with DCB# b (must be a constant, 0-9) into A$.

SET EOF

(For use with LDOS only). This statement allows you to truncate a random access file at a specified record. If you have a random access file (DCB 1 or 2 only, in BASIC/S) open, then to cause it to have 50 records (instead of say 100), just GET 1,R (where R=50) and then SET EOF1 (exactly as in LBASIC). Of course you need to close the file to make sure the directory entry is updated. This could also be done via PUT instead of GET; you just need to be positioned at the correct place in the file before you do the SET EOF. DO NOT try to SET EOF past the EOF - this bomb out with a DOS error.

CMD A$

Allows you to temporarily exit from your BASIC/S compiled code and execute a DOS command, and have control returned to your compiled program afterwards. Just set any simple string variable to the command you wish to execute, and then do a CMD A$. Be sure the command executed does not overwrite your code; compile your program starting at 7000H or higher to avoid this problem (or even at 56000!).

ON ERROR

A limited form of error trapping is possible with BASIC/S. In this form, you may trap for DOS errors only, not errors in BASIC or ROM processing. There is no ERL or RESUME in this form; all you can do is take some action based on the DOS error that occurred. First you establish your error trap routine with ON ERROR. Your ON ERROR statement MUST occur AFTER the line in your error trap routine you want to branch to; thus

```
50 ON ERROR GOTO 100
```

is no good since 100 comes later than 50. So your program would normally start out with a jump around the first line of your error trap, to your ON ERROR statement :

```
10 GOTO 40
20 A=ERR
```

```
30 GOTO 2000:'main error routine at 2000
40 ON ERROR GOTO 20
```

BASIC/S must already KNOW where in memory your error routine will be when it encounters the ON ERROR statement; hence the requirement for the error trap to come before ON ERROR.
The very FIRST thing your error trap must do is set some integer variable to ERR, to grab onto the error code. If you wait to do this, ERR will change and not be relevant. Finally, the code returned in ERR is the same as the DOS error codes that are explained in your DOS manual, which are returned in register A whenever a DOS error occurs. For example, if ERR were 24, this would indicate 'File not found'.

ASSIGNMENT statement :

Following are the allowed forms of the assignment statement.

INTEGERS :

Integer arithmetic is limited to +,-,*,/ and only 2 operands allowed on the right hand side. No builtin functions for integers. Constants may be used, however.

Thus:

```
X=A*B
X=5-B
```

Note that unary minus is not allowed here (for variables) ie X=-Y+Z is no good, while X=Z-Y is OK. However, with constants you may use unary minus freely. Anything of the form X=AsB is OK, where A and B are integer variables or constants and s is one of +,-,*,/, as long as you don't have two minus signs adjacent.

STRINGS

```
A$=B$
A$="constant"
A$=B$+C$      (simple concatenation)
```

Also we have the builtin string functions ASC, LEN, CHR$, LEFT$, VAL, RIGHT$, MID$, STR$, and INSTR. Where numeric arguments are required in the string functions, simple integer variables or constants must be used - no expressions. The actual string arguments cannot be constants, but:

```
A$=LEFT$(X$,2)
```

(for example) would be OK.

Also, expressions must be reduced to their simplest form -- e.g., concatenation within a function or function composition is not allowed. Break it down!

Note: The INSTR function differs from the regular DISK BASIC one in that no starting position may be specified -- syntax is just N=INSTR(A$,B$). However, unlike previous versions of BASIC/S, ALL of B$ is searched for, not just the first character.

MID$ note -- you can use MID$ on the left hand side of the = sign, and in that case, you can use either of the two forms MID$(A$,N)=B$ or MID$(A$,N,L)=B$ -- but they will give the same results, i.e. the length of B$ is used, L is ignored in the second form. If the source string (B$) is null, nothing is done.

Note III: The INKEY$ function is implemented, and must be used in the form: A$=INKEY$ (or B$, etc.).

HEX$

This is a hex conversion function, not supported by TRS-80 Disk Basic, but is supported under Microsoft BASIC-80 (and by their BASIC Compiler). BASIC/S II also supports it; what it does is to take an integer argument (variable or constant) and convert it to a hex string equivalent in value to the original integer. Thus

```
A$=HEX$(-1):PRINT A$
```

would print out "FFFF" (no quotes).

DISK I/O statements

Essentially, you have ten disk I/O buffers available for use (0-9), all of which may be used for sequential access, and two (1 and 2) of which may be used for random access. Here are the specifics :

OPEN

The OPEN statement is essentially that of disk BASIC, except that the filespec must be a string variable and not an expression in quotes. Syntax is

```
OPEN"m",b,F$<,r>
```

where m = mode = I,O,R, or E
b = buffer = (0-9) (constant only)
(must be 1 or 2 for direct access)
F$ = filespec (variable only)
r = logical record length (optional -- may be either an integer constant or an integer variable).

BASIC/S makes few restrictions on your use of the disk I/O statements, so be careful. For example, if you wanted to open a sequential file with an LRECL of 16, you could. However, you would probably be well advised to stick to direct access files for this!

OPEN"E" is like OPEN"O" except you start out positioned at the end of the file.

Sequential I/O is done with the LINE INPUT# and PRINT# statements. Just specify a buffer number adjacent to the #, and you are ready to go. Only a simple string variable may be input or output, although PRINT#1,A$; will disable the carriage return.

Random disk I/O is accomplished via the following :

FIELD

You must field your buffer in order to communicate between your strings and the disk file being accessed. Syntax is :

FIELD 1,nn AS A$,mm AS B$, ...

-- the buffer can be 1 or 2, the strings can be any of A$ thru Z$ (no array references allowed here!), and the numbers 'nn', 'mm' etc. must be integer constants (1-255 -- 0 is not allowed). Also you can't really use a multiple FIELD stmts for the same file -- the second will override the first. Moreover, the statements to process a random access file must be statically nested -- i.e. do not GOSUB or GOTO a later line to FIELD a buffer and then return to do your LSETs and PUTs, etc. Just OPEN the file, FIELD the buffer, process it, and CLOSE it, without GOSUBS and GOTOS. (At least, don't branch anywhere outside the range of statements between the OPEN and CLOSE stmts).

LSET

To place your strings into the buffer prior to being

PUT to the disk, use LSET. Thus

```
LSET A$=B$
```

where A$ is one of the strings mentioned in your FIELD statement. If LEN(B$) is less than that of the field variable A$, it will be filled out with spaces in the buffer. If greater, only the leftmost portion of B$ (for the fielding length of A$) will be in the buffer.

PUT

Syntax is PUT b,N where b is the buffer number (1 or 2) and N is any integer variable, containing the record number to be put. The record number variable is not optional.

GET

As in GET 1,R -- gets the Rth record from the disk file, and places its contents into the string variables mentioned in the FIELD statement.

LOF

The LOF function is implemented and syntax is

```
N=LOF(b)
```

where b is the buffer number (1 or 2 -- must be a constant). This returns the number of records in the currently open file with buffer b.

CVI and MKI$

For convenience in reading and writing integers from/to direct access files, these functions are implemented as in TRSDOS. In case you were mystified as to exactly what they did -- well, if the integer N has the 2 byte representation (L,H), then MKI$(N) is just CHR$(L)+CHR$(H). CVI just does the exact reverse. As with most BASIC/S functions, these may be used only with simple integer/string variables.

CLOSE

There is no global close in BASIC/S -- you must mention the buffer number. Thus,

CLOSE 5

would close the file with buffer number 5. If you close a file that isn't open, you will bomb out with 'FILE NOT OPEN'.

EOF

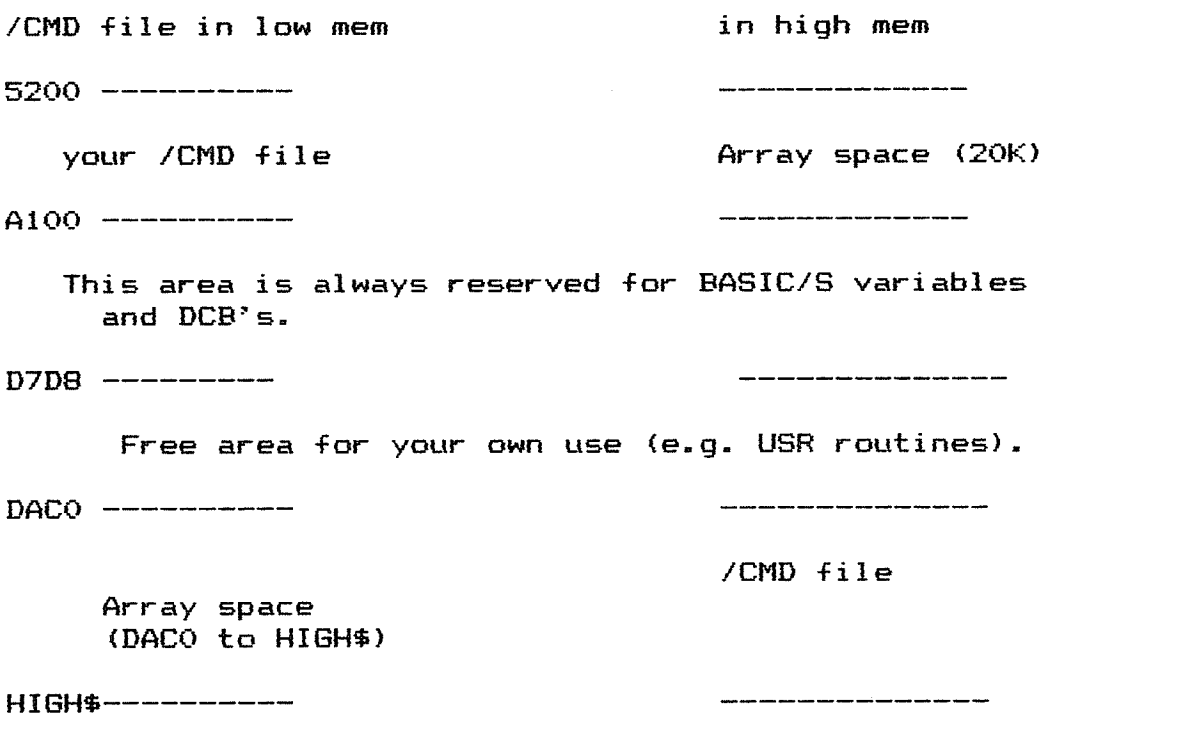
This isn't a function as such; it is to be used in a special form of the IF statement to check for EOF when inputting from a file. Simply say

IF EOF(b) THEN 200

(or whatever line number) to check for end of file on buffer b (0-9)

BASIC/S Memory Map

Following is a map of memory from 5200H up to HIGH$, showing how BASIC/S uses the memory in your TRS-80 (48K):



==========

--DISCLAIMER OF WARRANTIES & LIMITATIONS OF LIABILITIES --

We have taken great care in preparing this package. We make no expressed or implied warranty of any kind with regard to this manual or to BASICS/II. In NO event shall we be liable for incidental or consequential damage in connection with or arising out of the performance of this program.

Questions should be addressed to:

Bill Stockwell
4771 NW 24th #228N
Oklahoma City OK 73127
(405) 947-4156
Mnet 70070,320

Bill Stockwell may also be reached on the QSD Sig on MicroNet. Leave a message to 70001,610 for info or from the OK prompt, type R QSD<enter>.

Published by:
PowerSoft - a division of Breeze/QSD, Inc.
11500 Stemmons Expressway Suite 125
Dallas, Texas 75229
